North Carolina Statewide Technical Architecture

Application Domain

Table of Contents

1. Principles

1.1. Business drives application design.

Rationale:

- Information Technology (IT) exists to support the business through increasing efficiencies and maximizing economies. Avoid technology for technology sake.
- Make IT decisions, including application design and development, based on the needs of the business.
- Completely understand the business process by requiring each business event (triggers that activate business processes) to be identified and the work unit servicing each business event to be identified.
- Understand how work units cooperate to supply value when business events occur.
- Analyze existing processes to ensure alignment of the processes to the business goal.
- Draw logical boundaries around each process within the application.
- Implement each process with a collection of related business rules.

1.2. Continuous change in business and technology define a need for developing adaptable solutions.

Rationale:

- Solutions need to evolve over time to support new and changing business requirements.
- Adaptation may take different forms (modification of the structure, contents, or location of programs and data), and may be static or dynamic. Developing systems that can adapt to future needs will extend the life of applications.
- Adaptability of a system is reliant on a number of attributes or "ilities"
    - Extensibility - Staying flexible for future application expansion.
    - Scalability – Providing growth through the use of additional resources rather than extensive modification of the application itself.
    - Availability – Providing application readiness to handle customer requests and return timely and accurate responses.
    - Reliability – Providing resiliency and fault-tolerance of the application ensuring uptime and job completion.
    - Interoperability – The ability to exchange data between different applications utilizing differing technologies.
    - Accessibility – Ensuring open access of application functionality for persons with disabilities.
    - Manageability - Monitoring, tuning, and upgrading applications independently.
    - Usability – Providing an expected user experience and acceptance.
    - Mobility – Ensuring location transparency of the application or parts of the application.

1.3. Building a solution as a collection of reusable, loosely coupled components and services provide development efficiency and deployment flexibility.

Rationale:

- Applications designed with reusable components can be developed very rapidly, incur fewer costs, and reduce the risks accociated with new projects since the quality of the reusable component has already been validated.
- Details such as location of components, databases, and servers on which the components run are completely independent from the business process flows and definitions. Therefore this style of architecture is inherently adaptable, extensible, flexible, and scalable.
- Allow for the possibility of re-partitioning a deployed application in the future.
- Tightly coupled components become problematic when they limit the capability of the application, such as in development, testing, and deployment of the application. For example, a loosely coupled application provides flexibility in physical implementation (i.e., in the deployment of application components within a production environment).

1.4. An agency's architectural strategy defines the direction and requirements for development tool and framework selection.

Rationale:

- Do not rely on a single application tool to satisfy all development and application requirements.
- Most tools are oriented toward a specific area of development.
- Compile a suite of development tools with the ability to solve a variety of problems.
- Historically, the development tool selected directed project teams toward a specific architecture, which was supported by the tool. That led to the problem of the tools driving the architecture (and thus the business) rather than the business driving the architecture, which provides the requirements for the tools used.
- Select toolsets that align with the concepts provided within the architecture and do not limit the development team.
- Tools are now available for end-to-end design, development, and deployment of applications. Avoid a reliance on proprietary extensions and the default architectural guidance of these tools, which increase the likelyhood of vendor lock-in.

1.5. End users determine application usability and functionality.

Rationale:

- Design applications with the end user experience in mind.
- User experience includes the visual appearance, interactive behavior, and assistive capabilities of a software application.
- A focus on user experience is required from the beginning of the design process.
- Develop the user experience to be:

- Comfortable - familiar, friendly, and easy to use.
- Intuitive - very clear in explaining services and offerings. It should be clear how to find what is being looked for from a user's perspective.
- Consistent - maintain a set of standards and group similar items together.

1.6. The ease and completeness of testing is designed into the solution.

Rationale:

- Testing is a critical step in the development cycle.
- Trace test cases to documented, measurable, functional requirements.
- Design applications, components, and services to be easily tested and debugged.
- In addition to testing for potential bugs or defects, testing performance, fault-tolerance, and security are important elements of a test plan.
- Application components with consistent interfaces are easier to test on an application-wide basis.
- Error handling, tracing, and check pointing should be included, and implemented in the earliest phases of development.
- Incorporate testing into the earliest phases of the development process.

1.7. Proactive application management is essential for continuous service availability.

Rationale:

- Application management is a proactive, rather than reactive, process.
- Proactive application management better supports the business. With proactive management, applications report potential problem conditions at predefined thresholds before errors occur. This gives system administrators the opportunity to take corrective action to prevent an application from failing.
- While a solution can effectively rely on administrators to respond to errors in a reactive manner, proactive application management can be automatically undertaken using appropriate processes and management tools.
- Use thresholds to provide early alert to possible error conditions. For example, rather than having the database engine initiate an alarm when an application fails because its database table is full, a self-aware application alerts an operator when database growth meets some predefined threshold of maximum table size. This would allow system administrators to take corrective action in order to prevent a business-impacting outage.

1.8. Enterprise solutions exist in heterogeneous environments that necessitate interoperability.

Rationale:

- Interoperability is the ability of software located on different hardware, potentially from different vendors, to share data.

- While designing interoperable solutions is potentially more complex than designing for portability, the need for applications and services to be able to communicate with other services within a heterogeneous environment has increased.
- Component portability allows code to work on multiple platforms, but it requires that all software be written in the same language or limited number of languages. Interoperability, on the other hand, allows applications or services the ability to use or access other applications or services regardless of the language they are coded in or the platform they are deployed upon.

1.9. Ensure the confidentiality, integrity, and availability of application information.

Rationale:

- Security is an ever-growing concern, which is fundamentally about protecting the state's assets.
- Assets may be tangible items, such as documents or citizen information, or it may be less tangible, such as the reputation of the agency or state to provide needed services.
- Ensure that applications are developed in a secure manner by adhering to strict coding practices.

1.10. Utilize widely adopted standards for the development of applications and services.

Rationale:

- Standards reduce the complexity of application development by creating a homogeneous set of core, enabling, and transferable specifications.
- The use of standards provides improved product and process interoperability, and reduced cost of developing adaptable applications and services.
- Standards help ensure uniform, consistent, high-quality solutions.

1.11. Ensure critical systems maintain a clear vendor dependency risk mitigation strategy.

Rationale:

- Development projects that incorporate specific product technologies risk becoming dependent upon the vendor's implementation of that technology. When upgrades or software changes occur, interoperability problems may follow.
- Development teams must identify and plan for changes to vendor specific technologies.
- Complete dependence on a specific vendor's technology leads to a loss of control toward the technology, for the agency.
- Dependence on a specific vendor technology increases the responsibility of the developers. Developers must maintain an in-depth knowledge of the vendor's products and planned future product changes.
- Dependence can increase the costs of adapting solutions to changes in business needs, particularly if these business needs are inconsistent with the technology direction of the vendor.

1.12. Providing clear separation between functional layers enhances solution flexibility and adaptability.

Rationale:

- Splitting solutions along functional boundaries is consistent with the “separation-of-concerns” concept, which states that software should be decomposed in such a way that different aspects of the problem are addressed in well-separated modules or parts of the software system.
- Applications fundamentally consist of a minimum of three distinct layers, which include presentation, business rules, and data access.
- The separation of functional layers allows for the ability to adapt to changes.
- Functional separation provides a more scalable solution. As the transaction load, response time, or throughput requirements change, modules can be moved to other, more powerful systems or distributed over multiple platforms, without negatively impacting the service.

2. Application Design

2.1. Practices

2.1.1. Develop applications utilizing multi-tiered service-oriented approach.

Rationale:

- Mitigate the limitations and potencial reuse of monolithic and client/server design approaches by developing applications utilizing a multi-tiered destributed service-oriented approach.
- Multi-tiered destributed service-oriented applications are characterized by a functional decomposition of the application logic into process-centric services.
- Improved scalability, availability, manageability, and resource utilization can be achieved over other approaches.
- Applications created with this approach incure a lower total cost of ownership (TCO) because much of the application is built utilizing shared components and services. Additionally the ability of the application to adapt to changing business needs over time ensures longevity of the applications useful life.
- The maximum benefits of a service-oriented approach are realized when many applications are deployed across the state, sharing common software services and offering multiple user interfaces.
- To maximize sharing, reuse, and interoperability of common business processes across organizational boundaries, development teams should actively migrate their application design practices and tools to realize the benefits of modularized multi-tier designs and service-oriented computing.

2.1.2. Manage business logic outside of the development team.

Rationale:

- Manage an application’s business needs/requirements outside of the development staff.

- Assign responsibility for defining and maintaining the integrity of an application's business rules to business units.
- The development staff is responsible for coding and administering the software that implements business rules, while business subject matter experts (SME) manage the process requirements.
- Business units are responsible for the definition and integrity of business rules, and for communicating changes in business rules to the development team.

2.1.3. Incorporate the ability to capture and report management information within the application.

Rationale:

- Provide the ability to report status, performance statistics, errors, and other system conditions. Decide at design time which status events the application should report to users by function:
  - End Users (e.g., erroneous input)
  - Application managers (e.g., database growth within of maximum growth constraints.)
  - Network administrators (e.g., bandwidth utilization)
- Operations staff must be provided procedures for dealing with all conditions that are detected and reported. For example, if an application reports it can no longer access its database, operations staff must have instructions for handling the situation. These instructions should be documented in the agency Business Continuity Plan.
- At design time, decide the specific reporting requirements of an application module. Different applications may have different management needs, depending on their respective impact on the business the applications support.
- Applications are responsible for reporting conditions. Interpreting the reports and deciding on the appropriate response are performed external to the application, by management staff and/or automated within the management framework.
- Include run-time tracing to assist trouble shooting operational problems. Tracing should be able to be turned on and off by administrators.
- If no management environment exists, applications should still report status to local log files that can be monitored by administrators. Applications should still be able to read and respond to commands from administrators.

2.1.4. Develop applications to receive and process administrative commands.

Rationale:

- Decide at design time what control management tools exercise over the application components and services.
- Design applications to read and respond to commands from system administrators. Commands may include, but are not limited to, shut down and restart, reconfigure, and activation of audit logging/tracing.
- Make application configurations parameter-driven, so applications can be reconfigured without recompiling and redistributing code.

2.1.5. Utilize a standardized modeling language to align business requirements and application functionality.

Rationale:

- Utilize a modeling language to define “what” solutions and services are being developed in addition to “how” the development should be done.
- Identify business requirements and what processes the solution will contain to address the requirements.
- Solution modeling enables identification of opportunities for efficiency.
- Utilize a common, standardized modeling language such as the Unified Modeling Language (UML).
- Maintain consistent application models through the use of configuration management principles and best practices.
- Utilize a common and consistent modeling tool or language within the agency.

2.1.6. Publish open, standards-based interfaces for all reusable components and services.

Rationale:

- As the State continues to develop and publish reusable (shareable) application components and more course-grained, distinct business processes or services, the interfaces to both should be based on an industry-defined set of open standards. This provides the greatest industry-wide support, limits the potential for vendor lock-in, and reduces development complexity.
- Reuse is a key goal of component-based development and service-oriented architectures. Ease of reuse is the result of developing and designing open interfaces based on industry standards.
- Do not assume that application components are accessed via a specific application or user interface.

2.1.7. Utilize existing and proven architecture, design, and implementation patterns in the design of enterprise solutions.

Rationale:

- Patterns are logical design models of reusable ideas focused on abstract concepts such as extensibility, maintainability, scalability, reliability, and productivity.
- Many patterns are based on earlier designs that worked in practice and have gained industry support. Some have matured with widespread acceptance and are considered de facto standards in the industry such as the Model – View – Control (MVC) pattern, which can be used to separate business functionality from the presentation.
- Patterns provide roles, interactions, and relationships between application components and provide a solid foundation for building reusable components and services.
- Many development and modeling tools now provide support for patterns, enabling analysts or developers to apply patterns to designs. Note: While tools may support or push specific patterns, design decisions should be based on functional requirements of the solution and not the tool(s) used to develop the solution.

- Patterns should not replace developer knowledge; instead patterns should only be used as an aid to ease the development of common tasks and functionality.

2.1.8. Design components and services to be process-centric.

Rationale:

- Historically, application logic was developed as a set of program-centric components, which tended to be tightly coupled to the internal workings of a specific application or architecture. Process-centric components create a clear reference to the underlying business processes and sub-processes.
- The key goal of a process-centric environment is to define and clearly record the business processes (or services). This approach ensures that changes to business operational procedures can be added and assembled quickly by utilizing existing process flows.

2.1.9. Ensure accessibility by designing solutions that are available to the broadest possible range of users and compatible with a wide range of assistive technologies.

Rationale:

- As part of its goal to provide governmental services, the State must ensure that systems are accessible to persons with disabilities.
- Solution developers achieve accessibility by building applications that exhibit:
  - Device-independent design.
  - Universal design rather than custom views.
  - Equally usable for all.
  - Logical, portable, and clearly structured.
- Benefits of Accessible Designs include:
  - More usable application.
  - Easier and more cost effective to migrate or upgrade to take advantage of new technologies.
  - Consistent appearance and functionality across diverse computer configurations.
  - Access by all users regardless of rendering device used.

2.1.10. Limit Web Services development to internal, non-public facing initiatives, while industry standards mature.

Rationale:

- While many of the standards used in the development of Web Services are beginning to stabilize (e.g. WSDL and SOAP), critical pieces for manageability, security, reliable messaging, notification, and transactions are still relatively immature and volatile.
- New Web Service standards are being created, old standards are evolving, and vendors are integrating proposed standards into their products. This is creating a highly volatile environment for the development of secure, reliable Web Services.
- Disagreements among the participants within the standards bodies have also generated concerns about the direction of proposed specifications in areas such as orchestration, reliability, security, and transactions.

- Limit the use of XML-based Web Services to publishing services that are available to applications and other services statewide and within specific agencies such as for application separation, intra-agency integration, and inter-agency integration. Utilize current, mature distribution and integration technologies (application to application and messaging) for scenarios not listed here such as citizen to government or state to federal government applications.
- At this time Web Services are not recommended for use in interfacing with external government agencies (federal, local, etc.), partners, or providing services directly to the citizens based on the current state of standards. For these cases the use of current more mature technologies are favored.

2.1.11. Document the application design.

Rationale:

- The usefulness of an agency's application design is lost if not captured and documented.
- A documented design can be used as a training tool for new employees or consulting staff that is utilized for development and maintenance of applications based on the architecture.
- A properly documented design provides a common foundation for development teams to reduce errors and increase productivity.
- Application design supports business processes, and maintains the ability to change when business processes change. Changes occur more frequently in business processes than in the data required to support the business. Many of the cornerstones of application architecture and application development rely on the proper documentation of the agency's architecture and application design.

2.1.12. Design solutions that can be managed within a multi-tiered distributed environment.

Rationale:

- Develop applications and the components or services that interact with the application so they can be managed using the enterprise's system management tools.
- Manage the application as a whole by managing every component of the application as well as any component dependencies. Enable every component of the application to facilitate its own management.
- Application dependencies include infrastructure (e.g., middleware, databases, and networks), other applications, and shared software components. Application teams must identify and document these dependencies before an application is deployed.
- Typical management functions include:
  - Software distribution.
  - Start-up, shutdown, and restart of components and/or services.
  - Initialization of multiple instances of component (s).
  - Application Configuration.
  - Operations logging.
  - Notification of errors, exceptions, and unexpected events.
  - Security.

- Installation, removal, and update of application modules.
- Version control.

2.1.13. Design public facing, web-based solutions to be browser independent.

Rationale:

- Web solutions that target the public audience should refrain from utilizing browser specific features in the development of web-based UIs. Doing so limits the audience, reduces flexibility, and leads to vendor / platform dependancies.
- Construct web-based UIs with currently available standards (CSS, HTML4, and XHTML) to maximize cross-browser alignment.
- Solutions developed for a specific browser will inccur substantial switching costs (redevelopment of feature functions) if support for other browsers is needed in the future.

2.2. Standards

2.2.1. Comply with available application accessibility requirements.

Rationale:

- While each agency should adopt a level of accessibility that is consistent with the overall goals of the agency. As a foundation and in the absence of an agency accessibility level, all applications, web-based or standalone, are required to meet the minimum accessibility level.
- For web-based development:
  - The World Wide Web Consortium (W3C), an international standards body for such protocols as HTML, XML, and CSS, maintains the Web Content Accessibility Guideline. The accessibility guideline mirror Federal and International requirements for accessibility. The URL for the document can be found at http://www.w3.org.
  - “Minimum” for web-based applications is defined as full compliance of the World Wide Web Consortium’s Web Content Accessibility’s “basic” accessibility requirements (Priority 1 in version 1.0 or Level 1 in version 2.0).
  - Federally funded projects may need to comply with other standards such as the Federal Section 508 (http://www.section508.gov/).
- For standalone application development:
  - “Minimum” for standalone applications is defined as full compliance of the Federal Section 508, Subpart B, Software Applications and Operating Systems section (1194.21) (http://www.section508.gov/).
- While these standards present the minimally acceptable accessibility requirements, software developers are strongly encouraged to maximize the accessibility of their applications for universal access.
- Additional information can be found at the U.S. Access Board - The Access Board is an independent Federal agency devoted to accessibility for people with disabilities. http://www.access-board.gov/

2.2.2. Develop applications that can be managed by the Simple Network Management Protocol.

Rationale:

- The Simple Network Management Protocol (SNMP) is an application layer protocol that facilitates the exchange of management information between network devices.
- SNMP enables network administrators to manage network performance, find and solve network problems, and manage applications and services that are deployed on the network.
- The Internet Engineering Steering Group (IESG) maintains the SNMP specification (http://www.ietf.org/iesg.html).

3. Application Structure

3.1. Practices

3.1.1. Provide clear separation between behavior, content, and presentation of user interfaces.

Rationale:

- Provide user interface content without making assumptions about how it will be used.
- Accessible systems allow the client to process content in the form that best suits the client and user's capabilities.
- User Interface (UI) technologies continue to expand (interactive touch-screens, handwriting, voice recognition, and speech synthesis) with each technology utilizing a unique language such as WML, SpeechML, JSML, VoxML, HTML and XML.
- Reminder: Users of a system are trying to access information, not presentation.

3.1.2. Implement a discrete data access layer to isolate business logic from the persistence mechanism.

Rationale:

- Encapsulate in a data access layer the logic used to access data, such as querying or updating application data and logically separate it from the business layer.
- Initiate calls to the data access layer from within the business rules and not directly from the user interface or directly from other applications.
- Agencies are the custodians of the State’s data, and are responsible for maintaining the integrity of the data. By developing applications that allow only business rules to access data, the agency can maintain better control over data integrity.
- Data is created and used by business processes. In computer applications, data is created, used, and managed by the application component that automates the business process.
- Accessing data in any way other than by business processes bypass the rules of the module that controls the data. Data cannot be managed consistently if multiple processes or users access it directly.

3.1.3. Design solutions that can be physically separated across multiple environments.

Rationale:

- Physical tiers are divisions between the modules of your application, which may not map directly to the logical separation used to abstract the different kinds of functionality within the application.
- Physical separation does not necessarily equate to separate boxes or machines. Modules could be separated across virtual machines or within virtual partitions.
- The physical tiers may need to be separated by firewalls or other security boundaries in order to implement a layered security model.
- Physically distributing components over several tiers increases obstacles that potential attackers must circumvent in order to compromise the system.
- Distributing components across multiple physical tiers can improve an application's ability to provide continued scalability, availability, and manageability.
- Design physical separation into the solution early in the development lifecycle. Changes that need to be implemented after-the-fact incur unnecessary cost and time overruns.

3.1.4. Utilize standard distributed technologies for communication between application tiers.

Rationale:

- Standards-based communication is the goal (e.g., Web Services, XML messaging, ODBC, JDBC…), however the use of an industry accepted standards as it relates to a particular framework or environment is currently acceptable for non-service development (e.g., .Net Remoting, RMI, ADO.Net…).
- Do not develop or utilize a custom developed communication technology.
- Standards-based communication technologies do not always equate to a vendor-independent implementation. For example, the use of CORBA is not a guarantee of interoperability based on a specific vendor's implementation of the published specification. Development teams must be aware of this and mitigate the reliance on vendor specific implementations of published standards.
- XML-based Web services are an attractive alternative to communication between services. Additionally, XML-based Web Services are appropriate for use in integrating applications and services within the agency and between agencies. XML-based Web services are built on the common infrastructure of the HTTP protocol, XML, and SOAP. These are public, open-standards with wide industry support.
- Web services provide considerably more flexibility and interoperability for implementing solutions within a heterogeneous environment. However, due to their maturity level, the volatility of standards around this technology increases risk.

3.1.5. Develop applications and services independent of a particular deployment configuration or environment.

Rationale:

- Engage in logical application design separate from physical design.

- Do not focus on where application components will execute (i.e., where they will be deployed) or what specific production environment they will be executed within.
- Environments change over time and solutions will need to be reconfigured or redeployed based on the needs of the business.
- System designers and operations support staff make deployment decisions.
- Avoid building solutions that are unable to be deployed within a platform-neutral environment or contain hard-coded interfaces that are difficult to change.

4. Application Development

4.1. Practices

4.1.1. Utilize an enterprise framework in the development of applications and services.

Rationale:

- Utilize an enterprise framework that provides platforms, tools, and programming environments for developing multi-tiered distributed applications such as Java 2 Enterprise Edition (J2EE) and the .Net framework.
- Application frameworks provide an efficient, distinct, reusable, and unified software infrastructure that reduces the number of enterprise software products to support, maintain, and integrate.
- Application frameworks include such capabilities as presentation services, server-side processing, session management, business logic framework, application data caching, application logic, persistence, transactions, security, and logging services for applications.
- Application development tools and application servers are built on top of application frameworks.
- Leading frameworks are now providing support for the creation of Web services.
- Caution should be taken to implement applications with enterprise frameworks in a manner that avoids vendor lock-in.
- Standardization of an agency-wide enterprise framework encourages uniformity among development teams.

4.1.2. Define specific roles for developers.

Rationale:

- Separate development responsibilities along application boundaries.
- An application that has been properly separated provides developers an opportunity to specialize in the technologies and programming practices that support each layer.
- Developers with different specialties focus on areas of the application that best suit their skill sets.
- Separation of development staff might include:
  - User interface programmers.
  - Business rule programmers.
  - Data access programmers.
  - Application testers.

- Defining separate roles for developers based on the application boundaries limits the need for highly experienced generalists, which have mastered all aspects of application development. Less experienced developers can be used for focused development efforts.

4.1.3. Adopt and document a common set of coding standards.

Rationale:

- Document a set of common coding standards as part of the agency's architecture.
- The existence of a common set of coding standards easies the process of debugging and maintaining applications. They should address, but not be limited to:
  - Naming conventions for variables, constants, data types, procedures, and functions.
  - Code flow and indentation.
  - Error and exception detection and handling.
  - Source code organization, including the use of libraries and include files.
  - Source code documentation and comments.
  - Even the earliest code developed in a project should adhere to the standards.

4.1.4. Limit, identify, isolate, and document development activities that incur vendor dependencies.

Rationale:

- Identify and design a layer of abstraction to the underlying infrastructure or product-dependent software interfaces when possible.
- This isolation layer assures applications and services maintain interoperability and consistency for future technology changes and limits vendor lock-in.
- Isolating the dependencies limits the risks and costs associated with migrating to other technologies.
- Isolating the dependencies avoids obsolescence due to changes made by the vendor.
- Isolating the dependencies reduce the risks and costs of software upgrades.

4.1.5. Reuse existing components and services.

Rationale:

- Components are fine-grained encapsulated functionality to support common development efforts; while services are course-grained, process-centric, business functions utilized to carryout a specific task.
- Build applications by assembling and integrating existing components and services, rather than by creating or recreating common functionality.
- Components and services can exist internal to a group or agency or across agency boundaries.
- The objective of functional reuse is to decrease costs and time of developing systems.
- Reusable components increase the productivity of the application development departments within the enterprise.
- The use of proven components enhances the accuracy of information processing.

- As the State begins to move towards adoption of a services-oriented architecture (SOA) approach, development teams will be able to take advantage of component reuse within the application and the services provided by other applications.

4.1.6. Create and implement a comprehensive testing strategy as part of the solution’s development lifecycle.

Rationale:

- Software testing is a vital part of the development lifecycle.
- A testing strategy includes both white box and black box testing methods.
  - White Box - testing the internal structure of a program.
  - Black Box – testing how well a program meets its requirements, looking for missing or incorrect functionality.
- Testing plans contain, but are not limited to the following testing levels:
  - Unit Testing - testing specific functionality in isolation from the rest of the system.
  - Integration Testing - testing the interfaces between functional components or modules.
  - Function Testing – testing the required functions against specifications.
  - System Testing – testing system as a whole on platform similar to production.
  - Acceptance Testing - testing the completed solution by a small group of end users.
  - Regression Testing – re-testing solution after updates or changes have occurred.
  - Load testing – testing simulated real-life workload conditions for the application under test.
  - Latency Testing – testing the time taken by the client and the server to complete the execution of a request.
  - Scalability Testing – testing the capability to increase service capacity.
- No amount of testing can guarantee the application is free from defects, but a properly tested application can minimize the risk of functional failure.

5. Component Based Development

5.1. Practices

5.1.1. Document and publish information about reusable components.

Rationale:

- The reuse of the available components relies on their proper documentation.
- Maintain information about the available collection of reusable components.
- Changes will occur in the design, functionality, and interface of components over time.
- The usefulness and reuse of common functionality is lost if not captured and documented.
- Components that are not published and properly documented will potentially be rebuilt from scratch for each future development effort.

5.1.2. Provide a well-documented Application Programming Interface (API) for all reusable components.

Rationale:

- A published API is how components and applications will access and communicate with the encapsulated functionality.
- Documentation for the API should include input and output parameters, which parameters are required, which parameters are optional, and the lengths and types of the parameters.
- Once published, dependencies on the component should be assumed. Any changes to the underlying functionality must align with the expected usage.

5.1.3. Construct components as reusable, fine-grained, functional application building blocks.

Rationale:

- Highly granular components promote the goal of reusability for generic functionality within a component-based development strategy.
- Components alone do not solve business problems or automatically map themselves to business processes and transactions. They are generic "building blocks" that can be utilized to standardize application functionality.

5.1.4. Design components as discrete elements that function independently of others.

Rationale:

- Design components to function as a "black box" process.
- Provide the ability for reusable components to be replaced, causing minimal impact to the whole system by maintaining the same API.
- Design components to allow individual testing.
- Dependencies developed into component structures contain a risk of producing unintended results when new component changes are implemented.
- Generic components provide greater opportunity for reuse.

6. Service Oriented Architecture

6.1. Practices

6.1.1. Encapsulate business processes into well-defined, self-contained, course-grained services.

Rationale:

- Business processes are activities conducted in order to meet the goals of the agency.
- Implement services by grouping objects, components, and other finer-grained services together and expose their functionality through a course-grained facade or service interface.
- Express services in business terms and encapsulate the business functional flows that exist between organizational units, lines of business, or the state.

- Provide services, which can be dynamically invoked.

6.1.2. Provide interoperable access to published services.

Rationale:

- The greatest opportunity for reuse and collaboration comes from services that can be deployed within a heterogeneous environment, and interoperate with other services regardless of their implementation.
- The state currently maintains a large number of applications and services that have been developed over time using a variety of technologies. In order to meet changing business requirements, new services must be able to use and interact with existing and future state assets.
- Standards exist for developing interoperable services such as the Web Services Interoperability (WS-I) profile, which consists of a set of non-proprietary Web services specifications. (http://www.ws-i.org/)

6.1.3. Access services through standardized, platform-neutral, self-describing, well-structured, and extensible messages.

Rationale:

- Interactions between services are achieved through an exchange of messages. Develop messages utilizing a common vocabulary, which defines the structure of those messages. A standards-based technology, such as eXtensible Markup Language (XML), should be used in the development of these message structures.
- Services utilizing standard messages ensure that the service will interoperate with other applications and services.
- Some vendors support a set of proprietary extensions used to enhance the functionality of XML. Vendor specific extensions create dependencies and lead to vendor lock-in.

6.1.4. Separate the service interface from its implementation.

Rationale:

- Maintain a layer of abstraction between the service implementation and its interface to ensure that a modification to the service does not impact the applications that rely on it.
- Access services through a service facade or interface, which acts as a liaison between entities.
- The service interface must provide the ability to utilize the service without considering its internal design and content ("black box").
- Service, as opposed to component functionality, is designed to be coarse-grained.
- Provide service interface documentation, which includes the business operations they perform as well as the required input parameters, possible errors or exceptions, and results.

6.1.5. Describe services using a standard format.

Rationale:

- Provide descriptions for published services that describe what information is needed to utilize the service, how the service can be invoked, and where the service is located.
- Providing this information in standardized formats eases the integration and interoperability of the service with clients and development tools.
- Business experts who do not necessarily possess in-depth technical skills should easily understand the intent of the service through its description.
- The Web Services Description Language (WSDL) is a supported industry standard for describing the interfaces and services of applications over the web.

6.1.6. Publicize and discover services using standard service registries.

Rationale:

- Centralize service offerings, publish descriptions of services, and manage services within a repository.
- Assign a resource to each registry, which is responsible for maintaining the service.
- Because other applications and services will become dependant on the available services, provide a high level of availability and dependability for the registry.
- The registry provides:
  - Efficient access to available service interfaces.
  - Preserve the integrity of published service interfaces.
  - Encourage the discovery and reuse of common services
- Utilize a standard registry model such as Universal Description, Discovery, and Integration (UDDI).

6.1.7. Utilize standard protocols for exchanging messages and data between services.

Rationale:

- Historically, vendor specific protocols were used to communicate with application logic, which was commonly deployed behind firewalls. In order to provide access to the application logic, firewall exceptions were needed. In a large enterprise these firewall exceptions could be extensive based on the number of different technologies that were used. By utilizing a standard protocol, the need for “open” firewall ports are limited to a manageable set.
- A growing number of services are being developed, which are accessible over technologies such as the Hyper Text Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP), by encoding and wrapping the data captured as eXtensible Markup Language (XML) messages. This provides adaptable systems that can interact with a broader range of available services.

6.2. Standards

6.2.1. Promote web services interoperability by conforming to the Web Services Interoperability (WS-I) Basic Profile.

Rationale:

- The Basic Profile consists of implementation guidelines recommending how a set of core Web services specifications should be used together to develop interoperable Web services.
- The Basic Profile is not a single standard. The The Basic Profile provides companion guidelines, conventions, and best practices to promote interoperable implementations.
- The Web Services Interoperability (WS-I) maintains the The Basic Profile (http://www.ws-i.org/).

6.2.2. Transmit web services messages as Simple Object Access Protocol (SOAP) compliant structures.

Rationale:

- The Simple Object Access Protocol (SOAP) is an XML-based protocol, which can be utilized for exchanging structured and typed information between applications in a decentralized, distributed environment.
- SOAP has the ability to be bound to HTTP or SMTP.
- The World Wide Web Consortium (W3C) maintains the SOAP specification (http://www.w3.org/2000/xp/Group/).

6.2.3. Use the Web Services Description Language (WSDL) for describing available services.

Rationale:

- Web Services Description Language (WSDL) is an interface description language for describing the interfaces and services of Web applications at any endpoint.
- WSDL allows Web services applications to publish and discover the services, interfaces, methods, protocols, and procedures for communicating between endpoints.
- Theoretically WSDL can bind to any protocol or message, although the WSDL specification only specifies SOAP, HTTP GET/POST, and MIME.
- The World Wide Web Consortium (W3C) maintains the WSDL specification (http://www.w3.org/TR/wsdl).

6.2.4. Utilize the Universal, Description, Discovery and Integration (UDDI) standard for publication and discovery of web services.

Rationale:

- The Universal Description, Discovery, and Integration (UDDI) specification defines standards to classify services according to published attributes.
- Relationships between services and attribute types can be defined, and services can be discovered through attribute value queries.
- The Organization for the Advancement of Structured Information Standards (OASIS) has ratified the UDDI specification (http://www.uddi.org/).

6.2.5. Conform to the eXtensible Markup Language (XML) specification in the development of web service messages.

Rationale:

- The eXtensible Markup Language (XML) is a simple, very flexible, descriptive language for writing structured data, which can be used to store or transport data.

- XML can be used with network protocols like HTTP.
- The World Wide Web Consortium (W3C) maintains the XML specification (http://www.w3.org/XML/).

6.2.6. Use the XML Schema specification for defining the structure, content, and semantics of XML-based messages.

Rationale:

- Historically XML metadata was described using Document Type Definitions (DTD), which evolved into a de facto industry standard prior to 1999.
- In 1999 the World Wide Web Consortium (W3C), which was maintaining the standard, halted further development of DTD standards.
- On May 3, 2001 the W3C published the XML Schema specification, which is the preferred metadata definition language (http://www.w3.org/XML/Schema).
- The World Wide Web Consortium (W3C) maintains the XML Schema specification.

6.2.7. Invoke services over Hypertext Transfer Protocol (HTTP)

Rationale:

- The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems.
- HTTP is a generic, stateless protocol.
- HTTP runs on top of TCP (Transmission Control Protocol), which is a stream protocol on top of IP (Internet Protocol).
- The Internet Engineering Task Force (IETF) maintains the HTTP specification (http://www.ietf.org).

7. Application Security

7.1. Practices

7.1.1. Ensure application security at all layers within a solution.

Rationale:

- A benefit of properly separating functional units within an application is the ability to implement a broader security model that focuses on vulnerabilities at each layer.
- Secure application environments by:
  - Providing perimeter security, which includes firewalls, intrusion-detection systems and anti-virus filters. These technologies are use to keep malicious traffic off the network.
  - Integrating security between the perimeter and the application. Additionally, security must be utilized between each layer of the application.
- A layered security model limits the amount of data and logic that is available at each layer if a system is compromised.

7.1.2. Leverage industry standard secure coding practices in the development of applications and services.

Rationale:

- Development teams should actively apply a secure coding methodology that aligns with their agency's architecture, which provides development teams direction for addressing security issues. Examples include:
  - Security infrastructures provide firewalls, intrusion detection systems, authentication services, and other methods of securing application environments. A secure coding methodology helps ensure the security of an application or service regardless of the underlying infrastructure.
  - Compromised systems or transaction exceptions should not reveal information about sensitive data or executions. Code to reveal the absolute minimum necessary to the service requestor, while capturing the appropriate security information to a secure log.
  - Develop applications assuming the absolute minimum set of privileges needed for execution by a user or other application.
  - Never depend on obscurity for security. Attackers with malicious intent have steadily increased their knowledge of existing security technologies and common vulnerabilities. Identify all sensitive data and managed appropriately such as ensuring passwords are encrypted both inside application executables and across the transport layer.
  - All input arriving from a source external to the application's current trusted environment is assumed to be malicious, until proven otherwise. Input should be validated prior to engaging in a business process.

7.1.3. Limit risk by exposing only necessary, well-documented service interfaces.

Rationale:

- Limit access to encapsulated application logic to only the components necessary to complete a business process.
- Develop complete documentation, which defines the appropriate use of available services.
- Limiting the number of entry points into the service reduces the attack surface exposed.

8. Purchased and Licensed Software

8.1. Practices

8.1.1. Limit and isolate customizations to purchased or licensed software.

Rationale:

- Purchased software applications have the potential to represent a significant challenge in the management and implementation of application, network, security, and platform architectures.
- Isolating customizations from the purchased software, improves the ability to upgrade and move to new releases as required over time. As new releases or patches to the purchased software occur, the customization can then be reapplied.

- Fully document customizations according to agency information technology documentation policies and procedures.
- Develop a customization support strategy to mitigate the risk associated customizing purchased or licensed software.

8.1.2. Purchase, transfer, or license business systems that provide clear separation between the presentation logic, business logic, and data access.

Rationale:

- Purchase or use commercial-off-the-shelf (COTS) packages that, at a minimum, are developed utilizing a multi-tiered distributed architecture.
- The specialized, process-centric functional separation provided by services within a Service-Oriented Architecture (SOA) is desirable though not required for purchased or licensed applications.
- A multi-tiered distributed architecture provides flexibility in deployment.
- Communication between the tiers should occur only with the adjacent tier.
- In contrast to monolithic or client/server design models, a multi-tiered distributed design encapsulates the application's business logic and makes it available to other applications and presentation technologies.